universidade federal de pelotas
CDTec
centro de desenvolvimento tecnológico

computação
Universidade Federal de Pelotas

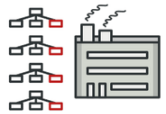
Programação Avançada
Hoje: Padrões de Projeto

Prof. Dr. Rafael P. Torchelsen
rafael.torchelsen@inf.ufpel.edu.br

Creational patterns

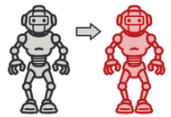
These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

Factory Method



Abstract Factory

Builder

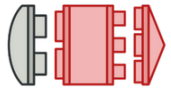


Prototype

Singleton

Structural patterns

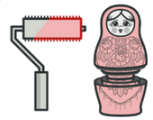
These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.



Adapter

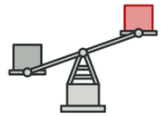
Bridge

Composite

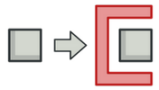


Decorator

Facade



Flyweight



Proxy

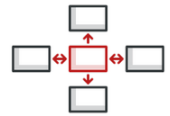
Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.

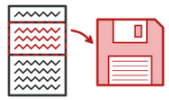
Chain of Responsibility

Command

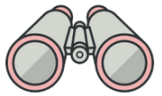
Iterator



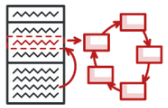
Mediator



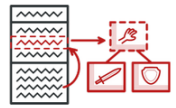
Memento



Observer



State



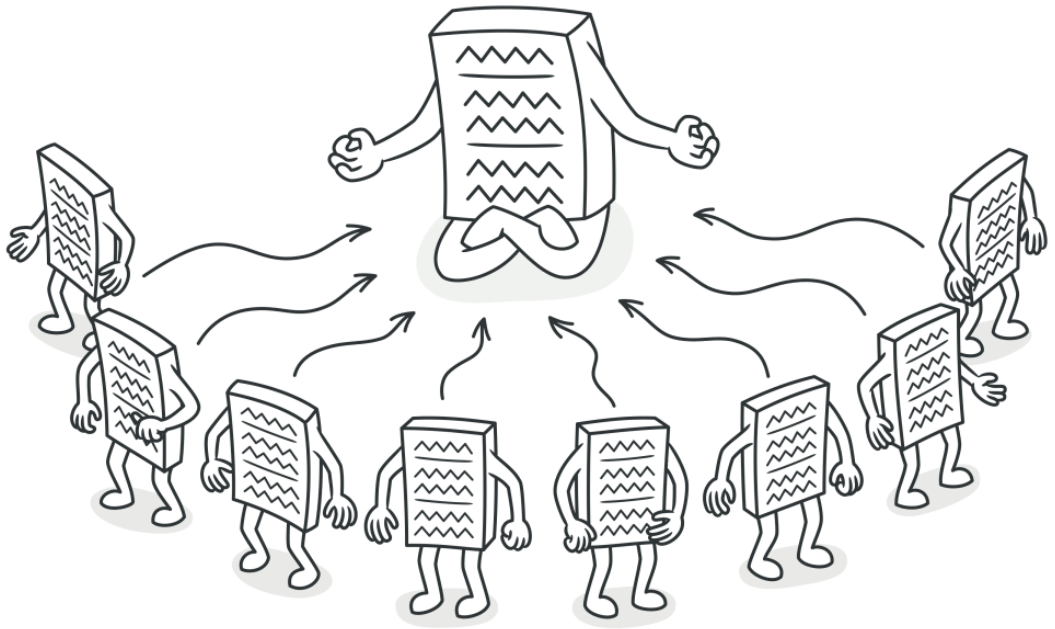
Strategy

Template Method

Visitor

Singleton

computação
Universidade Federal de Pelotas



Singleton é um padrão de projeto criacional que permite garantir que uma classe tenha apenas uma instância, ao mesmo tempo que fornece um ponto de acesso global a essa instância.

Problema

computação
Universidade Federal de Pelotas

Garantir que uma classe tenha apenas uma única instância.

Por que alguém desejaria controlar quantas instâncias uma classe possui? O motivo mais comum é controlar o acesso a algum recurso compartilhado — por exemplo, um banco de dados ou um arquivo.

Veja como funciona: imagine que você criou um objeto, mas depois de um tempo decidiu criar um novo. Em vez de receber um novo objeto, você receberá o que já criou. Observe que esse comportamento é impossível de implementar com um construtor comum, já que, por definição, uma chamada de construtor sempre deve retornar um novo objeto.

Problema

computação
Universidade Federal de Pelotas

Fornecer um ponto de acesso global a essa instância.

Lembra-se daquelas variáveis globais que você usava para armazenar objetos essenciais? Embora sejam muito úteis, também são muito inseguras, pois qualquer código pode potencialmente sobrescrever o conteúdo dessas variáveis e travar o aplicativo. Assim como uma variável global, o padrão Singleton permite acessar um objeto de qualquer lugar do programa. No entanto, ele também protege essa instância de ser sobrescrita por outro código.

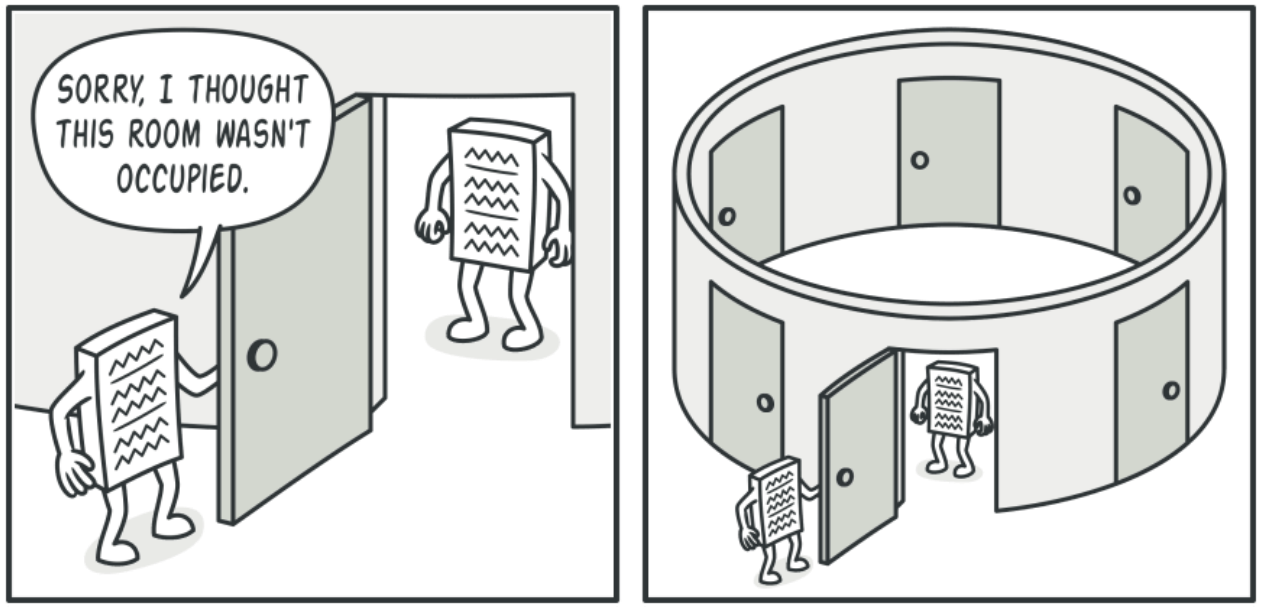
Há outro lado desse problema: você não quer que o código que resolve o problema nº 1 esteja espalhado por todo o seu programa. É muito melhor tê-lo dentro de uma única classe, especialmente se o restante do seu código já depender dele.

Solução

computação
Universidade Federal de Pelotas

Todas as implementações do Singleton têm estes dois passos em comum:

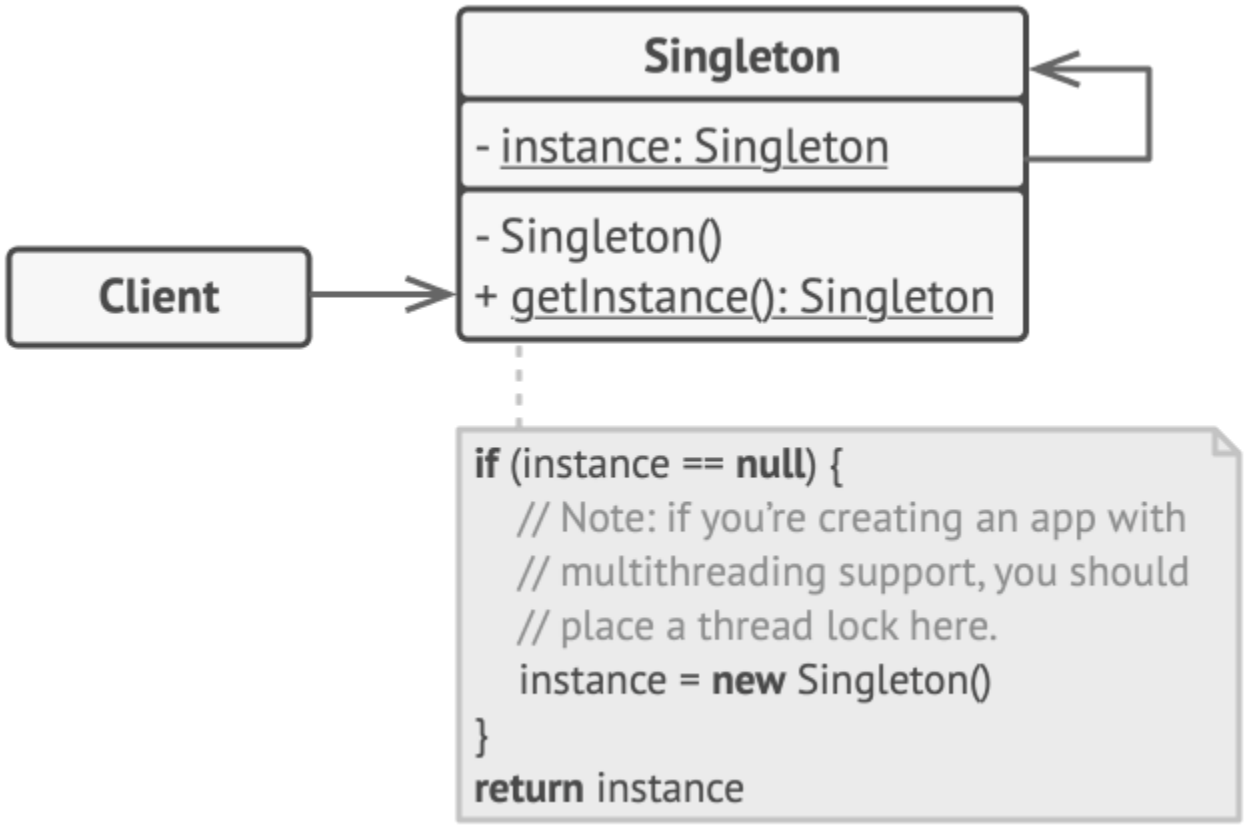
1. Tornar o construtor padrão privado, para impedir que outros objetos usem o operador `new` com a classe Singleton.
2. Criar um método de criação estático que atua como um construtor. Internamente, este método chama o construtor privado para criar um objeto e o salva em um campo estático. Todas as chamadas subsequentes a este método retornam o objeto em cache. Se o seu código tiver acesso à classe Singleton, ele poderá chamar o método estático do Singleton. Assim, sempre que esse método for chamado, o mesmo objeto será retornado.



Estrutura

computação
Universidade Federal de Pelotas

- A classe Singleton declara o método estático `getInstance` que retorna uma instância da própria classe.
- O construtor da classe Singleton deve ser ocultado do código do cliente.
- A chamada do método `getInstance` deve ser a única maneira de obter o objeto Singleton.



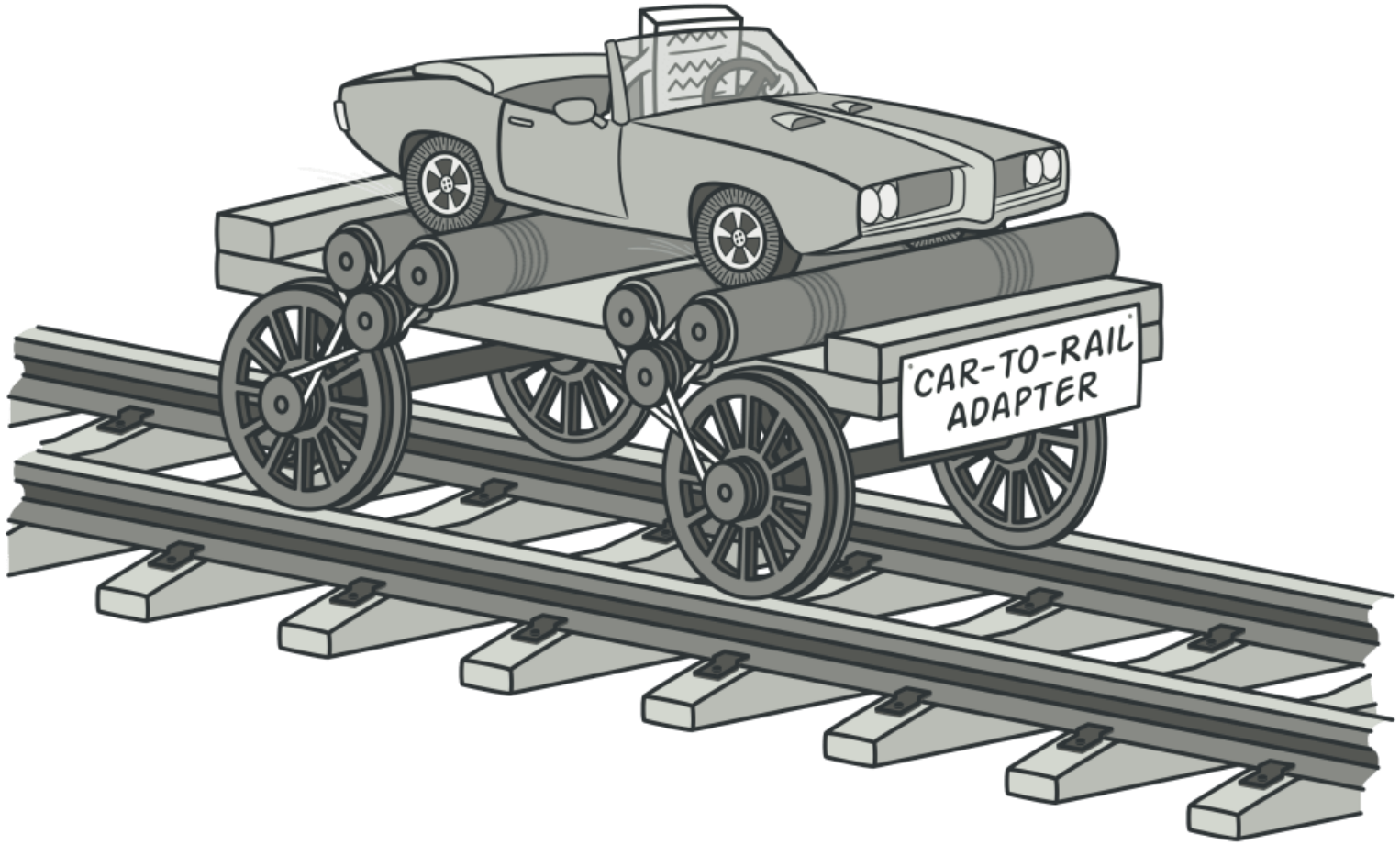
Pseudocode

```
// The Database class defines the `getInstance` method that lets
// clients access the same instance of a database connection
// throughout the program.
class Database is
    // The field for storing the singleton instance should be
    // declared static.
    private static field instance: Database

    // The singleton's constructor should always be private to
    // prevent direct construction calls with the `new`
    // operator.
    private constructor Database() is
        // Some initialization code, such as the actual
        // connection to a database server.
        // ...

    // The static method that controls access to the singleton
    // instance.
    public static method getInstance() is
        if (Database.instance == null) then
            acquireThreadLock() and then
                // Ensure that the instance hasn't yet been
                // initialized by another thread while this one
                // has been waiting for the lock's release.
                if (Database.instance == null) then
                    Database.instance = new Database()
        return Database.instance

    // Finally, any singleton should define some business logic
    // which can be executed on its instance.
    public method query(sql) is
        // For instance, all database queries of an app go
        // through this method. Therefore, you can place
        // throttling or caching logic here.
        // ...

class Application is
    method main() is
        Database foo = Database.getInstance()
        foo.query("SELECT ...")
        // ...
        Database bar = Database.getInstance()
        bar.query("SELECT ...")
        // The variable `bar` will contain the same object as
        // the variable `foo`.
```

computação
Universidade Federal de Pelotas

Adpter

computação
Universidade Federal de Pelotas



O Adapter é um padrão de projeto estrutural que permite que objetos com interfaces incompatíveis colaborem

Problema

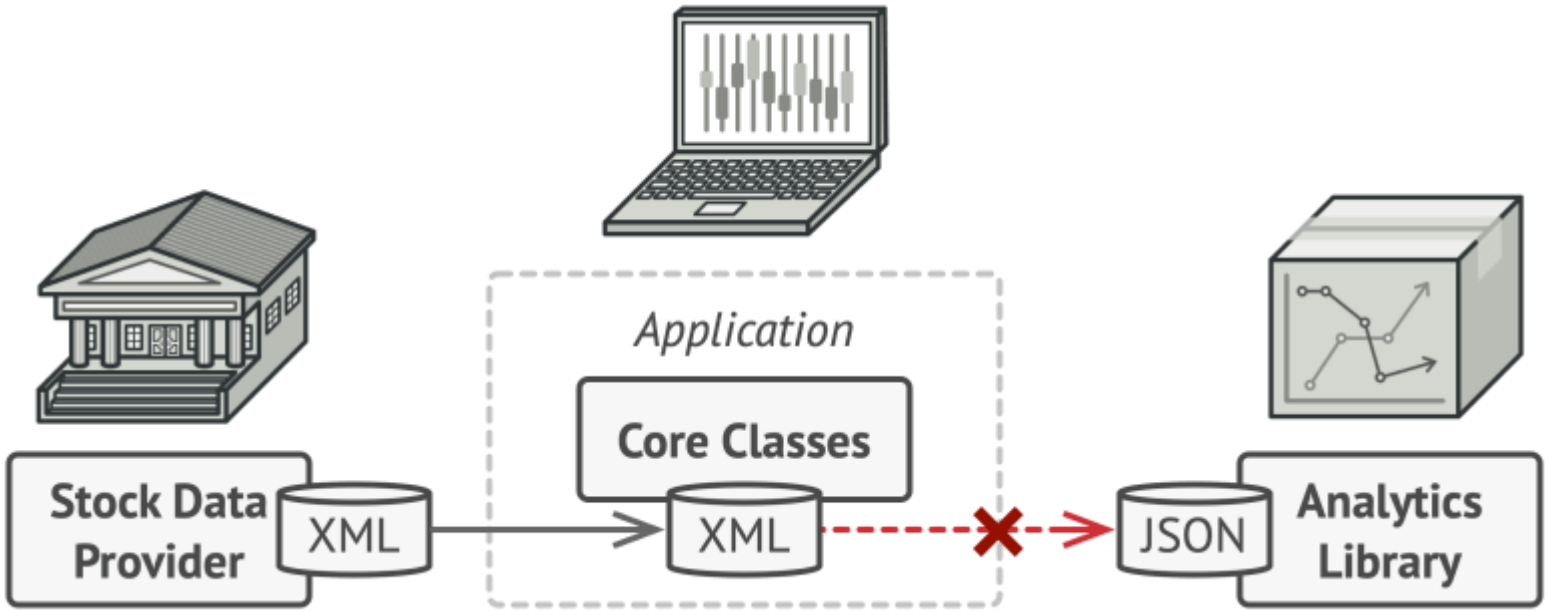
computação
Universidade Federal de Pelotas

- Imagine que você está criando um aplicativo de monitoramento do mercado de ações. O aplicativo baixa dados de ações de diversas fontes em formato XML e, em seguida, exibe gráficos e diagramas visualmente atraentes para o usuário.
- Em determinado momento, você decide aprimorar o aplicativo integrando uma biblioteca de análise inteligente de terceiros. **Mas há um porém**: a biblioteca de análise só funciona com dados em formato JSON.

Problema

computação
Universidade Federal de Pelotas

Você poderia modificar a biblioteca para funcionar com XML. No entanto, isso poderia quebrar algum código existente que depende da biblioteca. E pior, você pode nem ter acesso ao código-fonte da biblioteca, tornando essa abordagem inviável.



Solução

computação
Universidade Federal de Pelotas

- Você pode criar um adaptador. Este é um objeto especial que converte a interface de um objeto para que outro objeto possa interpretá-la.

- Um adaptador envolve um dos objetos para ocultar a complexidade da conversão que ocorre nos bastidores.

- **O objeto envolvido sequer tem conhecimento do adaptador**. Por exemplo, você pode envolver um objeto que opera em metros e quilômetros com um adaptador que converte todos os dados para unidades imperiais, como pés e milhas.

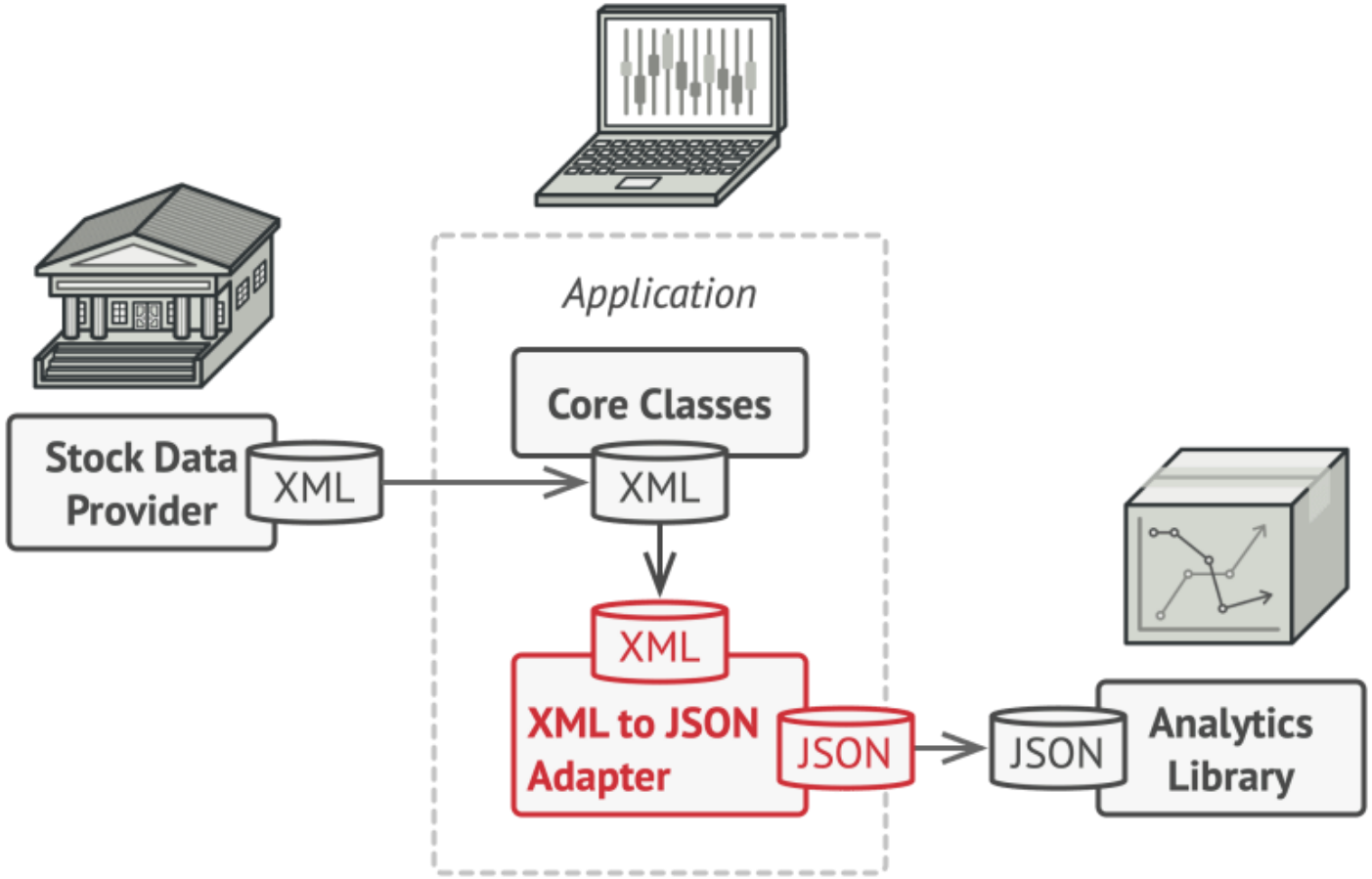
Solução

computação
Universidade Federal de Pelotas

- Os adaptadores não só convertem dados em vários formatos, como também ajudam objetos com interfaces diferentes a colaborarem. Veja como funciona:
  - O adaptador recebe uma interface compatível com um dos objetos existentes.
  - Usando essa interface, o objeto existente pode chamar os métodos do adaptador com segurança.
  - Ao receber uma chamada, o adaptador passa a solicitação para o segundo objeto, mas em um formato e ordem que o segundo objeto espera.

Solução

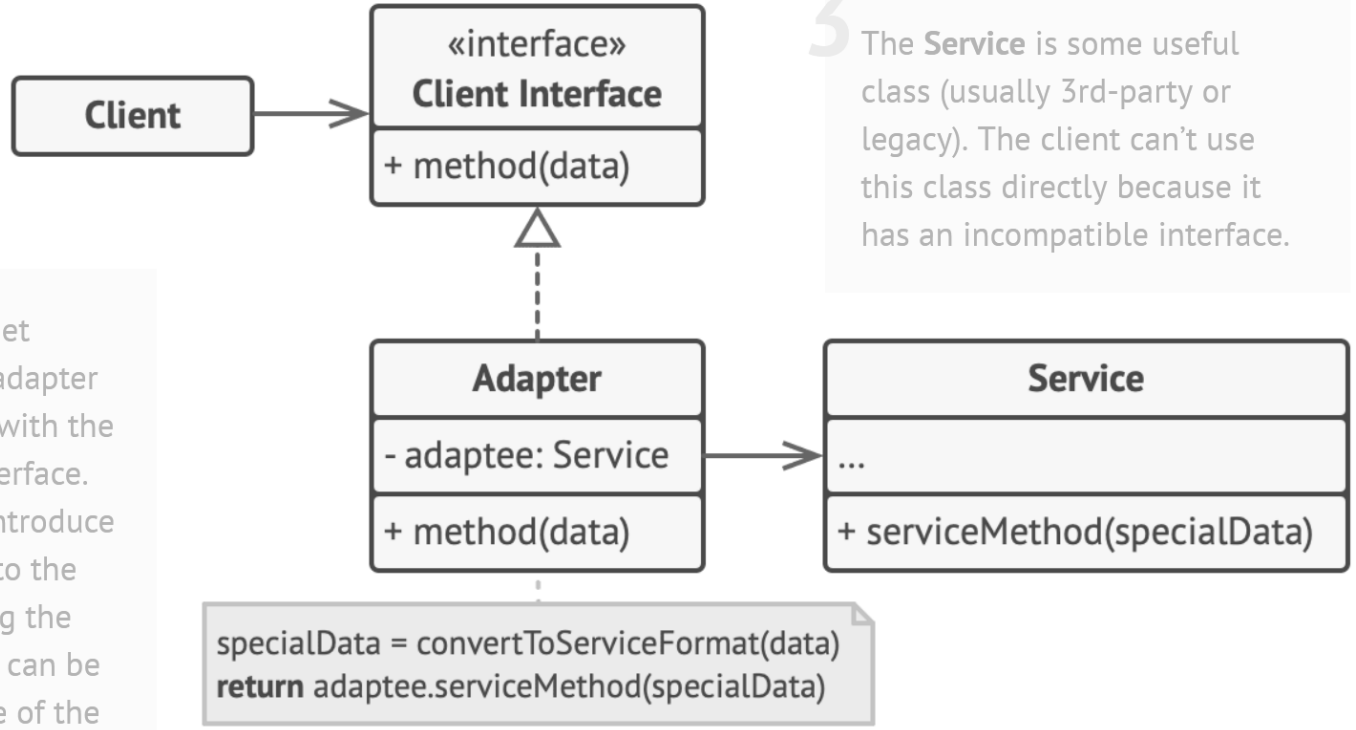
computação
Universidade Federal de Pelotas



Estrutura

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

1 The **Client** is a class that contains the existing business logic of the program.

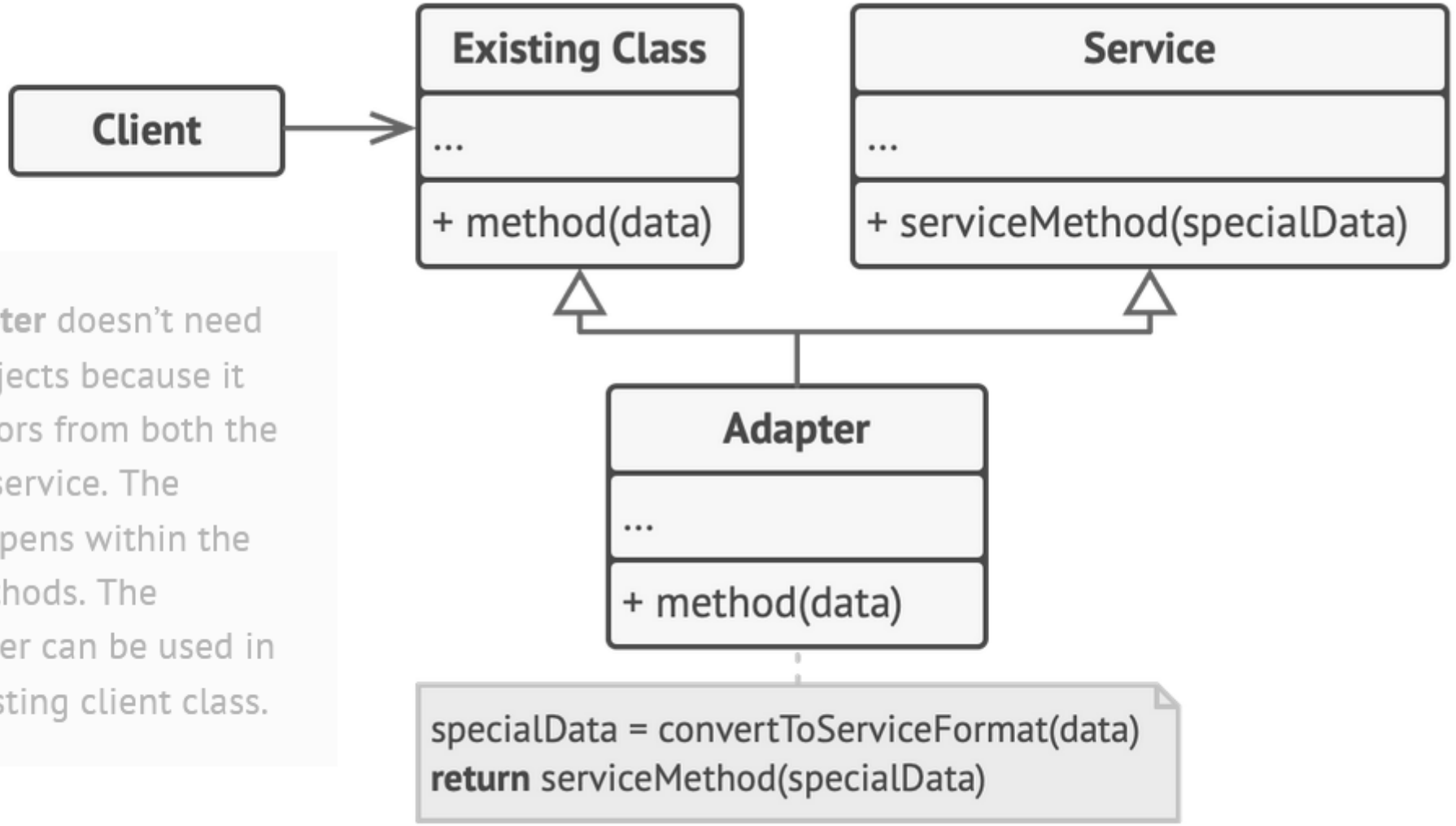


3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

Estrutura

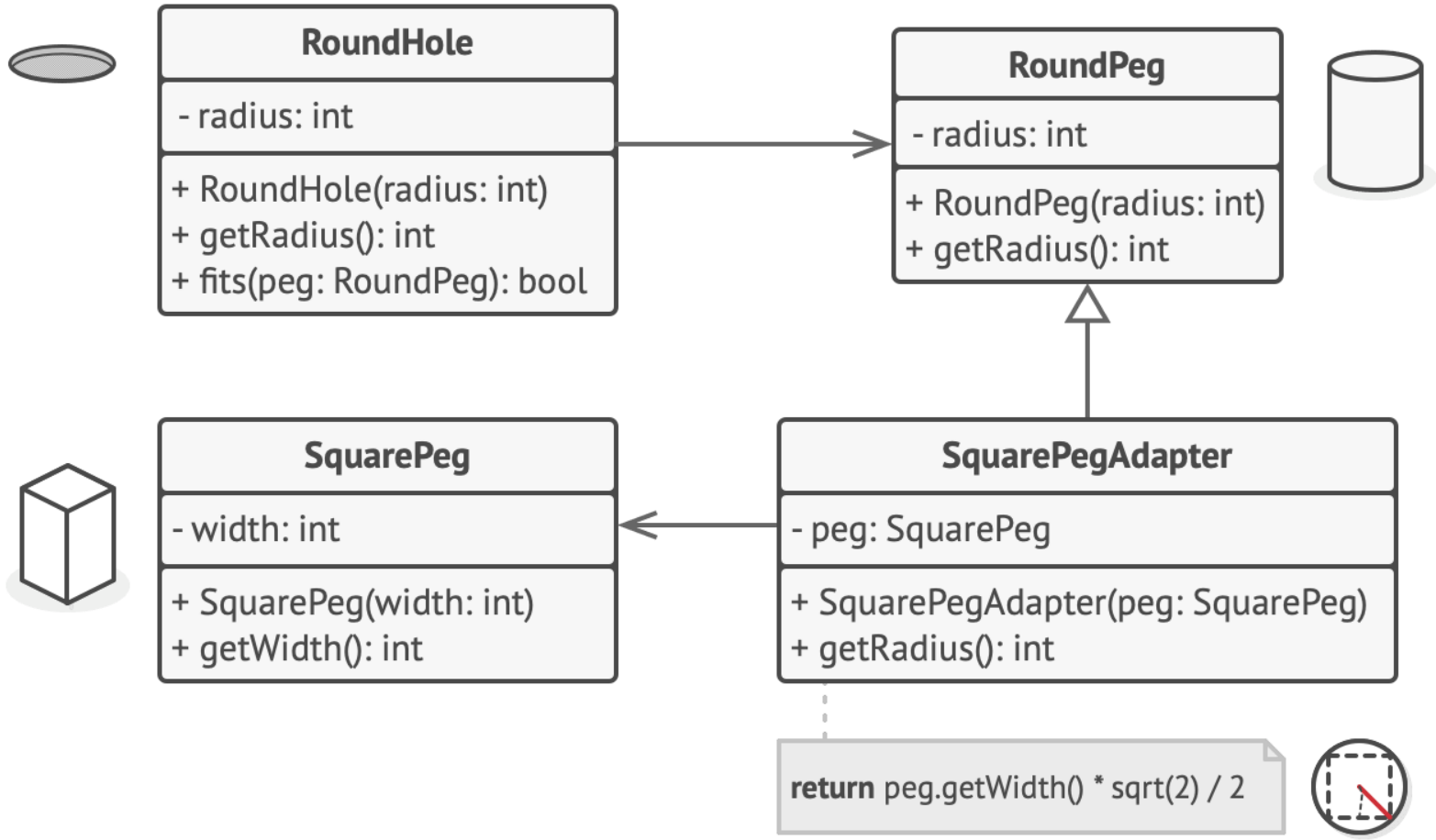


1 The **Class Adapter** doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

Esta implementação utiliza herança: o adaptador herda interfaces de ambos os objetos simultaneamente. Observe que essa abordagem só pode ser implementada em linguagens de programação que suportam herança múltipla, como C++.

Exemplo

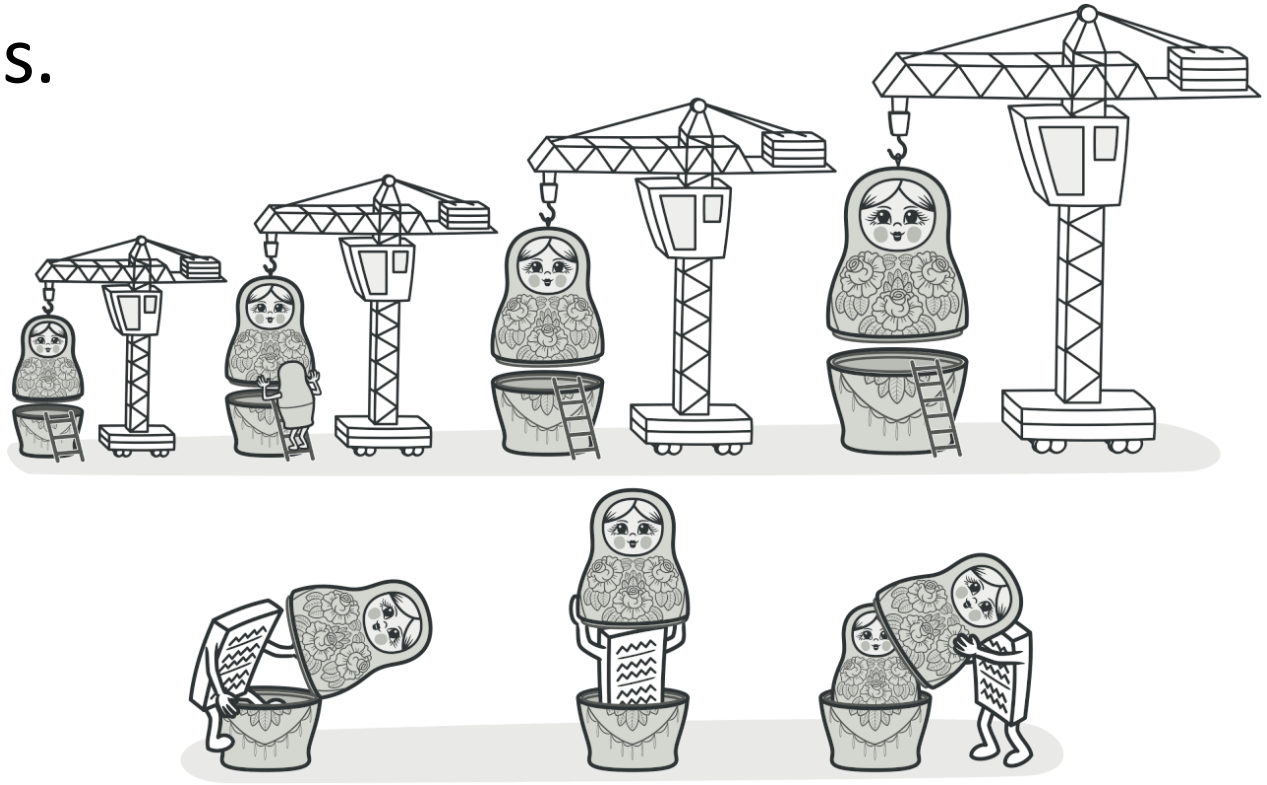
computação
Universidade Federal de Pelotas



Decorator

computação
Universidade Federal de Pelotas

- Decorator é um padrão de projeto estrutural que permite **associar novos comportamentos a objetos**, colocando esses objetos dentro de objetos encapsuladores especiais que contêm os comportamentos.



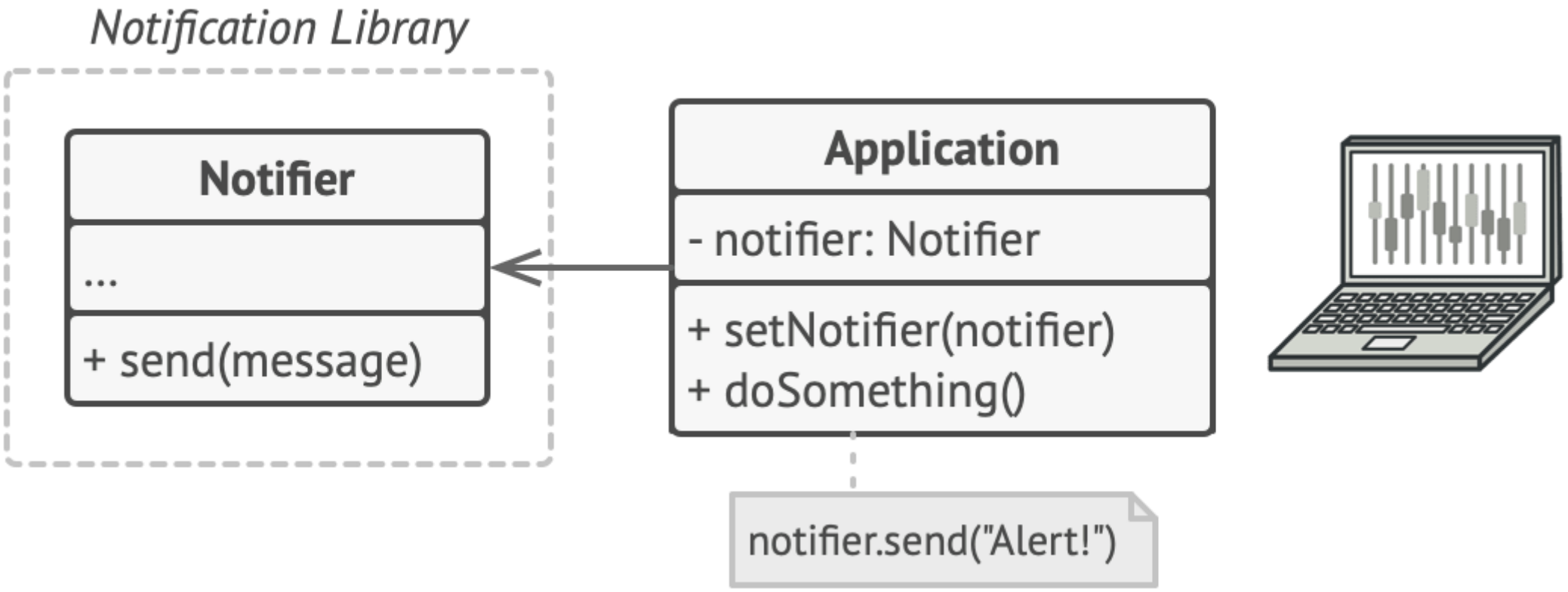
Problema

computação
Universidade Federal de Pelotas

- Imagine que você está trabalhando em uma biblioteca de notificações que permite que outros programas notifiquem seus usuários sobre eventos importantes.
- A versão inicial da biblioteca era baseada na classe `Notifier`, que possuía apenas alguns campos, um construtor e um único método `send`.
- Esse método podia receber uma mensagem como argumento de um cliente e enviá-la para uma lista de e-mails que eram passados para o notificador através de seu construtor. Um aplicativo de terceiros, que atuava como cliente, deveria criar e configurar o objeto notificador uma única vez e, em seguida, utilizá-lo sempre que algo importante acontecesse.

Problema

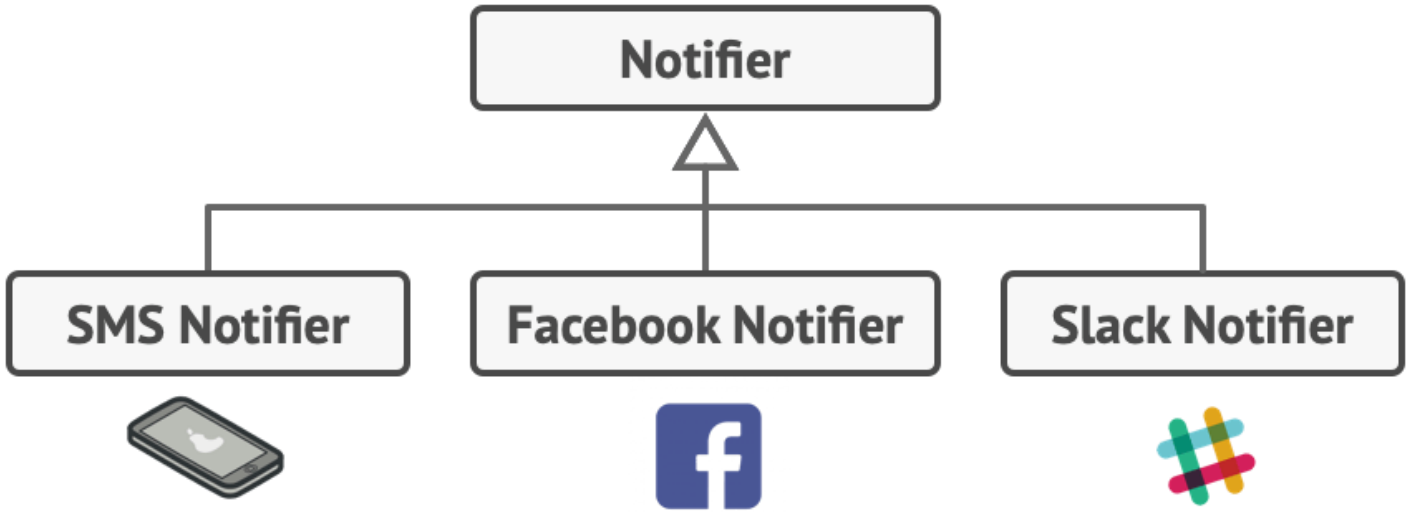
computação
Universidade Federal de Pelotas



Problema

computação
Universidade Federal de Pelotas

- Em algum momento, você percebe que os usuários da biblioteca esperam mais do que apenas notificações por e-mail. Muitos gostariam de receber um SMS sobre assuntos críticos. Outros gostariam de ser notificados pelo Facebook e, claro, os usuários corporativos adorariam receber notificações pelo Slack.



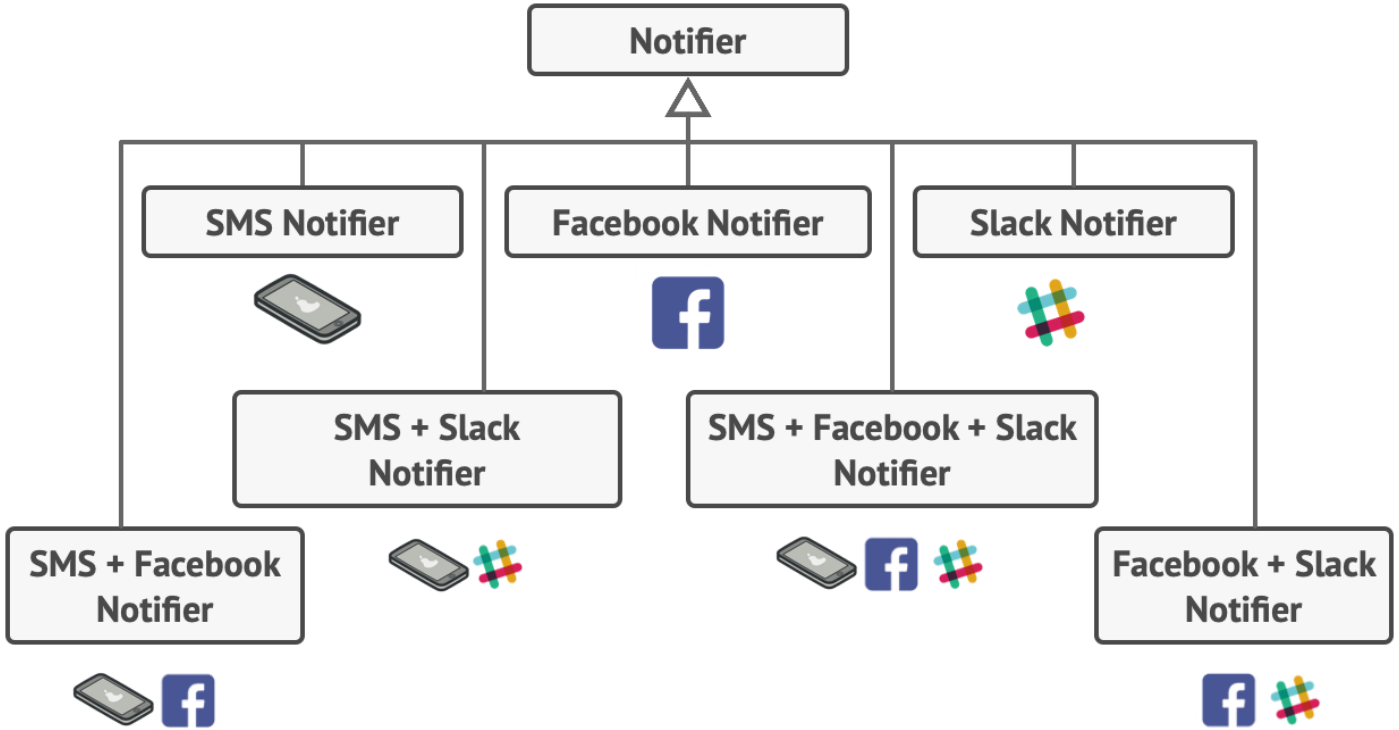
Problema

computação
Universidade Federal de Pelotas

Quão difícil pode ser isso? Você estendeu a classe Notifier e colocou os métodos de notificação adicionais em novas subclasses.

Agora, o cliente deveria instanciar a classe de notificação desejada e usá-la para todas as notificações subsequentes. **Mas então alguém**, com razão, perguntou: "Por que você não pode usar vários tipos de notificação ao mesmo tempo?

Se sua casa estiver pegando fogo, você provavelmente gostaria de ser informado por todos os canais."



Solução

computação
Universidade Federal de Pelotas

- Estender uma classe é a primeira coisa que vem à mente quando você precisa alterar o comportamento de um objeto. No entanto, a herança tem várias ressalvas importantes que você precisa conhecer.
  - A herança é estática. Você não pode alterar o comportamento de um objeto existente em tempo de execução. Você só pode substituir o objeto inteiro por outro criado a partir de uma subclasse diferente.
  - As subclasses podem ter apenas uma classe pai. Na maioria das linguagens, a herança não permite que uma classe herde comportamentos de várias classes ao mesmo tempo.

Solução

computação
Universidade Federal de Pelotas

Uma das maneiras de superar essas limitações é usando Agregação ou Composição em vez de Herança.

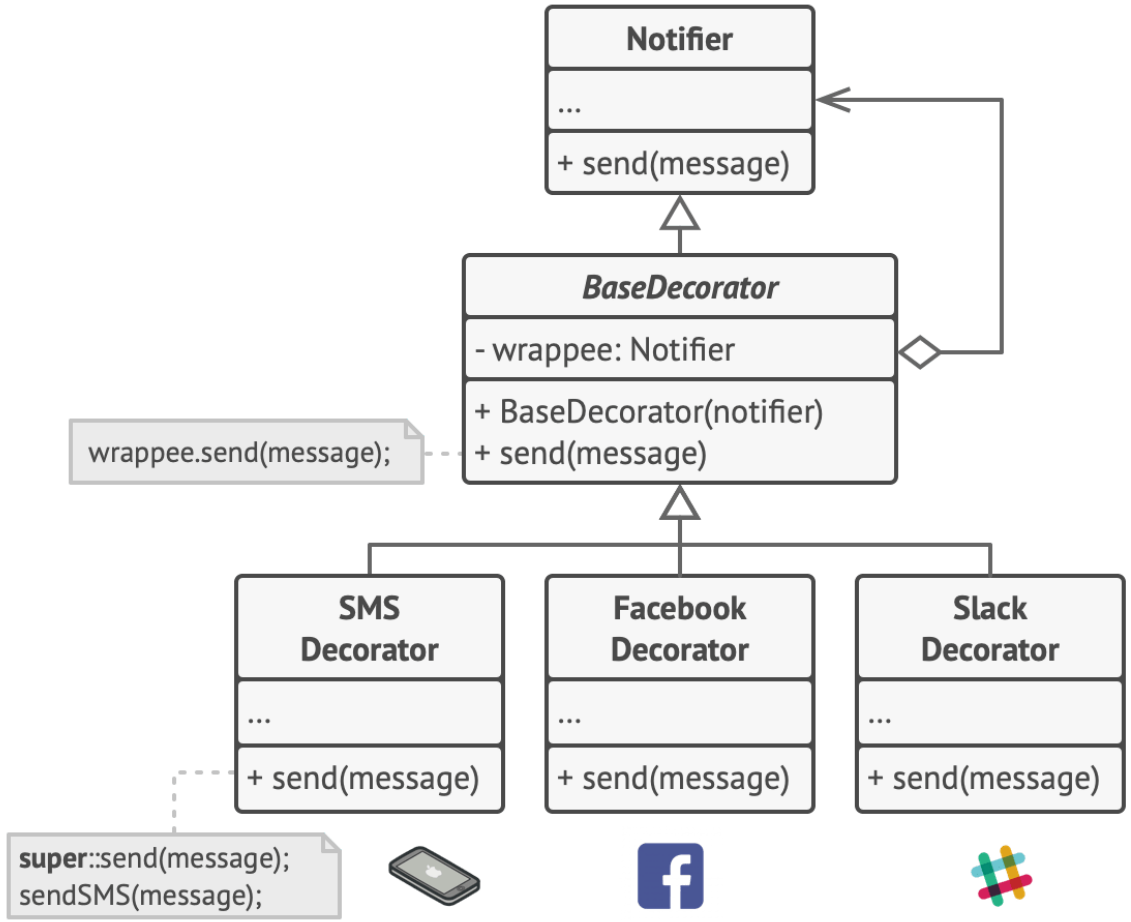
Ambas as alternativas funcionam de maneira quase idêntica: um objeto possui uma referência a outro e delega a ele algumas tarefas, enquanto que, com a herança, o próprio objeto é capaz de realizar essas tarefas, herdando o comportamento de sua superclasse.

Com essa nova abordagem, você pode facilmente substituir o objeto "auxiliar" vinculado por outro, alterando o comportamento do contêiner **em tempo de execução**. Um objeto pode usar o comportamento de várias classes, possuindo referências a múltiplos objetos e delegando a eles todos os tipos de tarefas.

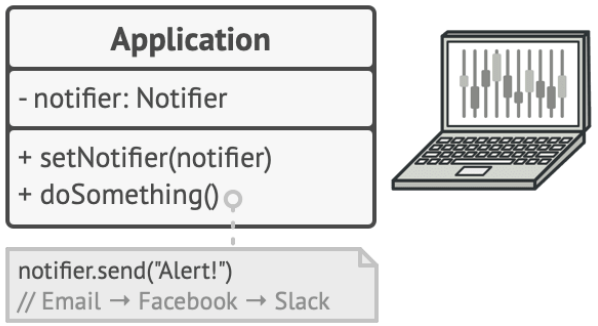
Solução

computação
Universidade Federal de Pelotas

Em nosso exemplo de notificações, vamos manter o comportamento simples de notificação por e-mail dentro da classe base Notifier, mas transformar todos os outros métodos de notificação em decoradores.

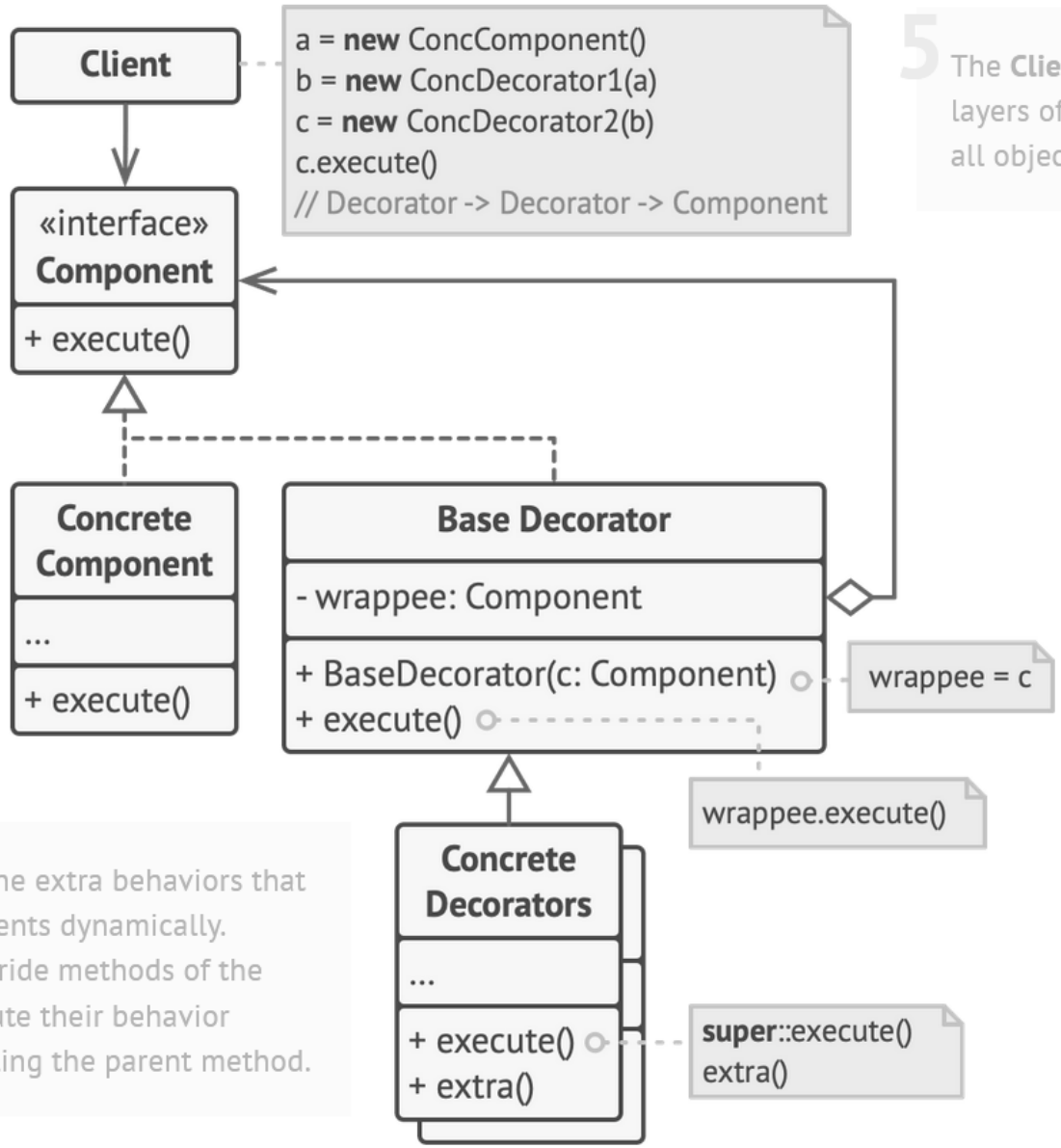



```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```




```
notifier.send("Alert!")
// Email → Facebook → Slack
```

Estrutura



1 The **Component** declares the common interface for both wrappers and wrapped objects.

2 **Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.

3 The **Base Decorator** class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.

4 **Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.

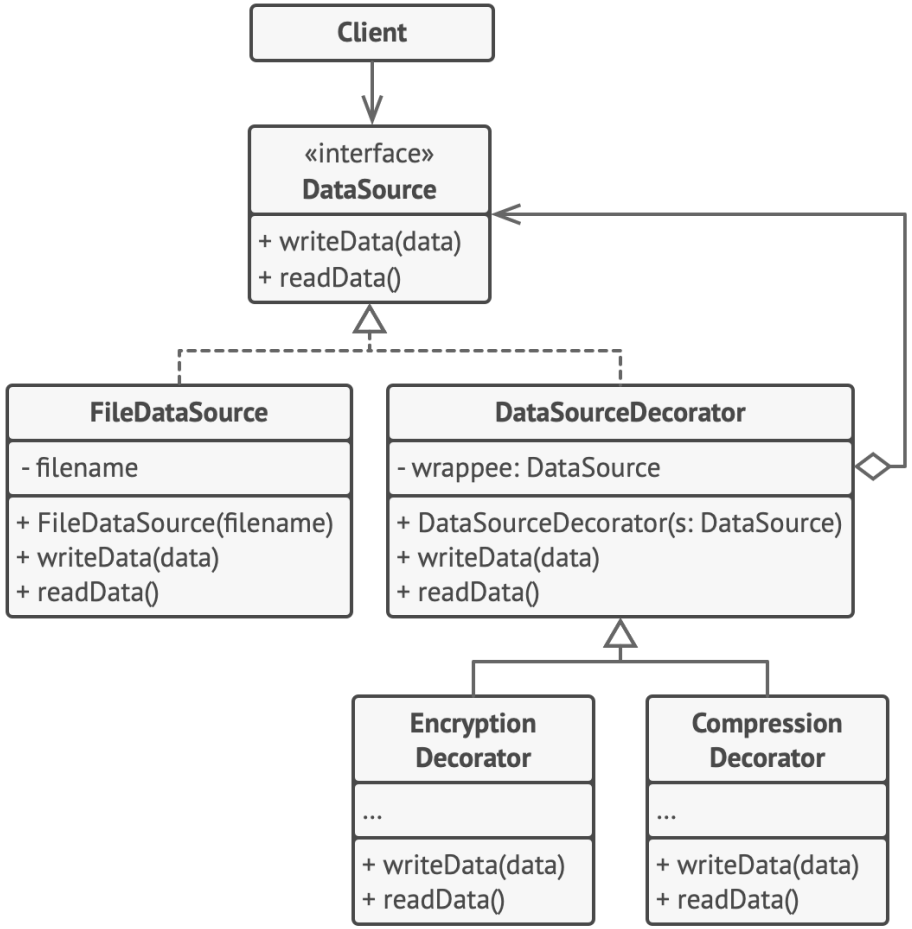
5 The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

Pseudocode

computação
Universidade Federal de Pelotas

O padrão Adapter oferece uma interface completamente diferente para acessar um objeto existente. Por outro lado, com o padrão Decorator, a interface permanece a mesma ou é estendida. Além disso, o Decorator suporta composição recursiva, o que não é possível ao usar o Adapter.

Utilize o padrão Decorator quando precisar atribuir comportamentos extras a objetos em tempo de execução sem quebrar o código que utiliza esses objetos.



Neste exemplo, o padrão Decorator permite comprimir e criptografar dados sensíveis independentemente do código que realmente utiliza esses dados.

Exemplos práticos

computação
Universidade Federal de Pelotas

https://refactoring.guru/design-patterns/examples

Trabalho 7: Padrão de projeto (continuação)

computação
Universidade Federal de Pelotas

- **Continuar o trabalho 6 incluindo novos padrões**
  - Use o SINGLETON para os elementos que só existe 1 durante todo o tempo, por exemplo, o grid ou gerenciador de caminhos
  - Use o ADPTER para permitir grid retangular e hexagonal, por exemplo, pegar todos as células vizinhas de outra célula não funciona igual em ambos, mas com o ADPTER a interface é a mesma
  - Use DECORATOR para agregar diferentes comportamentos/algoritmos para os agentes e obstáculos em tempo-real

- **GIT**
  - Código
- **Prazo**: 25/11 23:59